A RESOURCE AWARE SOFTWARE ARCHITECTURE FEATURING DEVICE SYNCHRONIZATION AND FAULT TOLERANCE

Chris Mattmann
University of Southern California
University Park Campus, Los Angeles, CA 90007
mattmann@usc.edu

Bilal Shaw
University of Southern California
University Park Campus, Los Angeles, CA 90007
bilalsha@usc.edu

ABSTRACT

We present a component-based software architecture that dynamically discovers and consumes remote services from distributed devices connected across a network. The architecture maintains its own local functionality, while also actively participating in its environment by discovering and responding to other devices as well. One novel capability of this software is its ability to synchronize its local and remote services with all other devices in its environment via its meta-architecture infrastructure. Furthermore, our architecture is fault tolerant and has the capability of re-synchronizing with lost connections and remembering old peers. The software architecture is built on top of the PRISM middleware and inherits much of its design style from the C-2 Architectural style.

We have deployed our extensions to PRISM, and subsequent software architecture on a network of distributed devices that included Windows 2000 Pentium III-based computers, and wireless Compaq IPAQ PDAs and have created a sample application, a distributed calculator, as a proof of concept for our extensions.

KEYWORDS

Fault Tolerance, Resource Allocation, PRISM, C-2.

1. INTRODUCTION

The need for distributed software systems has reached an all-time high. Needs such as distributed databases to encourage data sharing across networks, and embedded systems that need persistent communications, storage, and availability has forced modern computer scientists to investigate the capability of software systems to function in environments that are not normally considered as “good” hosts for computer systems. Hardware limitations such as slow processors, limited memory, and unreliable network connections have created a need for functionality such as *fault tolerance*, *resource awareness*, and *resource management* within these constrained environments.

These needs (as well as others) are being addressed by the PRISM [1] architecture. The PRISM architecture extends the idea of topological component systems that exchange messages and contain no shared address space by introducing peer-to-peer components and communications that span distributed and embedded systems. It also includes support for critical embedded capabilities including ***asynchrony, efficiency, delivery guarantees***, and ***disconnected operation***. PRISM has been applied to a variety of domains including an education tool in a graduate-level software engineering course, a distributed military battlefield simulation [2], and a distributed map visualization and navigation application [2].

Our work extends PRISM to add meta-architecture support for connectivity (and *disconnectivity*), as well as threaded device synchronization and fault-tolerance for disconnected operation at the meta-architecture level (through storage of previous stable connections that are no longer existent in the environment).
We demonstrate this work with a distributed calculator system. The application that we built was a piece of java software that runs in a distributed and loosely coupled environment. It was implemented with Sun's *Java Development Kit, version 1.1.8.* Because of the system independent nature of the Java Virtual Machine, our application worked on a variety of platforms including a *COMPAQ IPAQ PDA* System running *Windows CE*, and several different Windows machines running *Windows 2000*, with JDK 1.1.8 for Windows installed.

Section 2 begins by briefly introducing the C-2 Architectural style, and the PRISM middleware. It also discusses what meta-architectural information is, and how it can be used to provide transparent functionality to a distributed system. Section 2 also discusses our implemented extensions to PRISM. Section 3 presents a distributed calculator that we tested across a network of Windows PC's, running Windows 2000, and 2 networked Compaq IPAQ PDAs.

2. PRISM MIDDLEWARE AND C-2 ARCHITECTURAL STYLE

The C-2 Architectural style specifies a topology of components, and connections between them. C-2 applications are built from message passing components with no shared address space that share *connectors* between them. Connectors propagate 2 types of messages between their connected components:

1) A *Request* is a message that travels **up** the architecture that requests some type of functionality from the components that receive the message.
2) A *Notification* is a message that travels **down** the architecture that returns the results from the requested functionalities of the components.

Originally, C-2 was created to handle the needs in graphical user interface applications [4], where modularity and component interactions aid in the ability for development of re-useable software toolkits. C-2 was shown to have novel properties in other domains as well, including domains in which heterogeneous components are developed in different languages (C++,Java,Perl,etc.) across a variety of platforms (Linux, Windows, etc), and need to be *composed* into a single communicative system with desired properties and functionality.

The PRISM framework extended the C-2 architectural style to support development of applications in distributed, mobile, and resource-constrained environments. PRISM adds peer to peer component messaging capability through *peer connectors*, and *peer messages.* PRISM also adds the ability to handle the **disconnected operation**, and a class of connectors called *Border Connectors* that communicate across networks to C-2 in order to facilitate its application to the aforementioned environments.

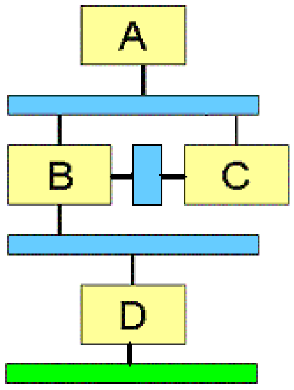

Figure 1. **A Sample Prism architecture with 4 components, labeled A,B,C,and D. The components are connected via two local connectors. Note that B and C are connected by a single *PeerConnector*. Also note that D is connected to a *BottomBorderConnector*, which spans device boundaries.**

Our work builds on the PRISM foundation and adds the meta-architecture through the implementation of the *Meta Component* which manages the *Border Connectors'* connection information, in addition to device services. All architectures that use our extension must have this *Meta Component*. The *Meta Component* manages local and remote device information such as *Provided*

Services, Resource Locators, and *Network Throughput*. We also introduce a *SynchDeviceThread*, which actively pursues lost connection information.

2.1 Meta-architecture information

One novel property of PRISM is its ability to handle the **disconnected operation**. PRISM reconnects disconnected devices by checking in a time-increment to see if a device is still connected. If the device does not respond, a *Reconnect Thread* is instantiated and tasked with attempting to reconnect to the disconnected device. If the disconnected device comes back online within a certain interval of reconnection attempts the *Reconnect Thread* reconnects the device, else the device is assumed to be fatally disconnected, and no further reconnections are attempted. It is important to note that this functionality is implemented at the architecture level, and all implementing software applications are equipped with this native functionality.

Even when the device is reconnected, PRISM assigns the software application the task of being aware that the device is back and actively participating in its environment, and more importantly that its provided-services are once again available for consumption. One way around this problem is to store some *meta-architecture* information about each device at the *architectural level*, so that all implementing software applications keep stored information about other devices that were in existence at some point during its life cycle. This information could consist of nothing more than a unique identifier for all known devices, and could be as detailed as remembering the provided-services of every device, device descriptors, device name, creator, etc. This *meta-information*, which is normally considered to be the burden of the implementing an application is a good candidate for being part of the architecture itself. This way when devices that are disconnected return to the environment, their functionality can be seamlessly re-integrated without any knowledge by the implementing application.

Table 1. Example meta-architecture information about a device connection

Device ID	Device Location	Provides Encryption Service?
D1	**127.99.92.225**	Yes
D2	**127.0.0.1**	Yes
D3	**26.93.195.222**	No

We propose a ***Meta Component*** to store the meta-architecture information that all implementing PRISM software applications need to define. The Meta Component contains three tables of information:

1) A *Services Table*, which contains {deviceID,service$_1$, service$_2$, …service$_n$} tuples. The tuples map devices to provided services
2) A *Devices Table*, which contains {deviceID, Device Location. M$_1$,M$_2$ ….M$_n$} tuples (where M$_1$..M$_N$ is a vector of meta-architecture data that would be useful to store for each device. Examples of this are CPU Speed and Network Throughput). The tuples map devices to device meta-data.
3) A *Disconnected Devices Table*, which contains information about disconnected devices. The information stored in this table is the union of the *Services Table and the Devices Table*.

The Meta Component is also responsible for filtering messages within the local architecture because of its unique positioning between two connectors.

2.2 Device Synchronization

The addition of meta-architecture information helps us to seamlessly re-integrate disconnected devices into our system at the architecture level; however, it does not provide the low level functionality of actually marshalling the disconnected device metadata to the Meta Component for re-entry into the *Devices* table.

We handle this problem by introducing another architectural level component, the *SynchDeviceThread*, which works in the following fashion. As each system is running, its own SynchDeviceThread will ping all devices in the *Device Table* every *n* seconds (where n is specified as a start-time parameter to the system). After the remote device is pinged, it is inserted into a **"soon to be disconnected"** queue in the *MetaComponent*. The pinged device then has 5 seconds to reply to the ping, in order to be removed from the disconnection queue. Whoever replied before 5 seconds is up and is removed from the disconnection queue. After 5 seconds all devices that have not replied, will be removed from the *Services Table* and the *Devices Table* and will be inserted into the *Disconnected Devices* table. This way, it is possible to reconnect with the device at a latter time when it is back in range (if it ever does come back in range). In essence, if a device comes into range and is pinged by the local calculator system, and it replies to the ping, then the local calculator system will check to see if the device that replied is in the current *Device Table*. If it is, then it must have been replying to an earlier ping, so it will be removed from the temporary disconnection queue if it is present there. If the device that replied is not in the *Device Table*, then it was possible that it was connected to the current device at a previous time, so we check the *Disconnected Device Table*. If it is in there, then its metadata is restored to the *Device Table and the Services Table.* The SynchDeviceThread also handles the case where a new device comes into the environment that has not yet been pinged—in this case it merely adds the device to the *Devices Table*.

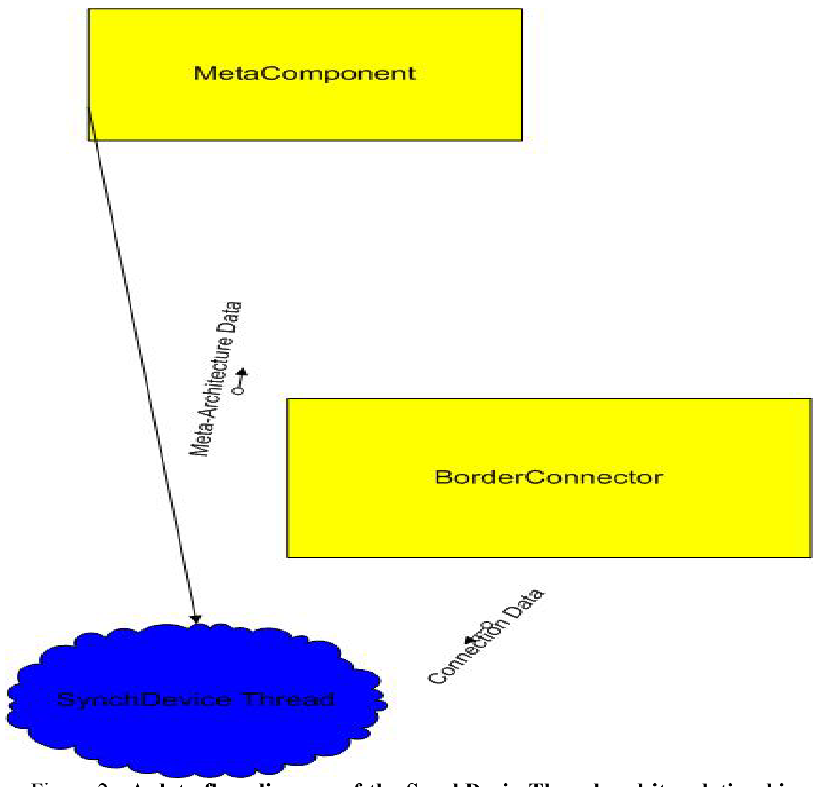

Figure 2. **A data-flow diagram of the SynchDeviceThread and its relationship with the Meta Component and the native PRISM architecture. The SynchDeviceThread receives connection data from BorderConnector, which consists of PING REPLIES. The SynchDeviceThread then conditionally dictates control over the MetaComponent's meta-architecture tables.**

3. A DISTRIBUTED CALCULATOR EXAMPLE

As a proof of concept of our additions to PRISM, we implemented a distributed calculator application that was deployed across a network with 2 COMPAQ IPAQ systems, and a set of Windows computers.
This project was part of ***Cs599: Software Engineering for Embedded Systems***, taught by Neno Medvidovic, at the **University of Southern California's *Center for Software Engineering (CSE)*.**

Our application runs in two modes:

1) A local calculator that relies on its own local math services to provide functionality to a GUI, where a user can enter 2 numbers into a text box window, and then click on a Button to perform a mathematical operation. The appropriate local *Math service* (on our system these service buttons are colored "red") will respond to the request and send the answer to the GUI, where a result will be displayed in the appropriate result box.
2) A calculator that can "consume" services from other calculators that it discovers dynamically in its relative world (given a bootstrap of known hosts, and service ports). The calculator provides a set of local services, but can also query other calculators. It can find out which services they provide, and if any services exist that are not available locally to its system, consume those services and abstract away the fact that they were really remote, and not locally available.

3.1 Graphical User Interface

We tried to keep the interface of the calculator very simple. Essentially we have provided sixteen different mathematical functions on our calculator. We felt that it would be very easy to enhance our calculator with scientific functions once we had the basic calculator functioning in a distributed environment. Our calculator performs unary and binary mathematical operations on *real* numbers and returns the result of the computation (again a real number) in another field. We have also provided the following buttons where each button performs a specific function. These buttons are: *Delete*, *Close*, *Connect*, *Disconnec, and Dump Device Data*. When the calculator first starts some of the buttons are colored red and some are discolored. The red color on a button indicated that a mathematical function is local to the calculator. The discolored buttons are non-functional. We can specify which services ought to be local and which ones remote based on a configuration file. When the calculator finds remote services on a remote host it will try to access them and the non-functional buttons will be appropriately colored blue, giving an indication to the user that he/she can access and use the remote services.

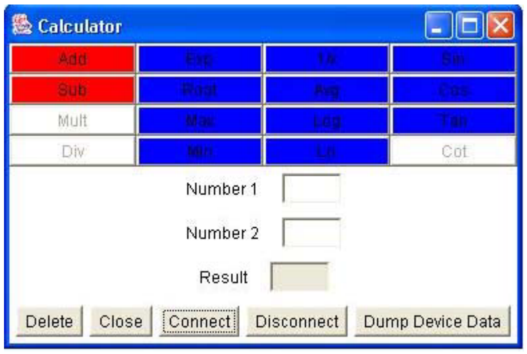

Figure 3. **The look and feel of the GUI Component of our application. Notice the red and blue colored buttons. Red buttons indicate locally provided services, and blue buttons indicate remote service availability.**

3.1.1 GUI Button Functionality

Each of the buttons in the GUI provide a different function. These functions are listed below.

- Math Buttons (‘Add’, ‘Sub’, etc): The math buttons allow the user to perform a mathematical function on 2 real numbers given in the *Number1* and *Number2* text inputs.
- Delete: The delete button clears the number fields and the result obtained from the previous calculation.
- Connect: This button allows a calculator to connect to any remote calculator in its environment. Upon successful connection to a remote device(s) the non-functional/discolored buttons on the local calculator turn blue, indicating to the user that remote services from the connected device(s) are now available.
- Disconnect: This button allows a local calculator to disconnect from all the remote calculators that it is currently connected to.
- Dump Device Data: This allows us to dump the data on the local device.

3.2 System Architecture

We used a Java based reference implementation of the PRISM framework provided to us by Neno Medvidovic. This implementation was tested against the *JDK 1.1.8* in order to ensure compatibility across all devices that were deployed with the calculator application (mainly the 1.1.8 version was used to allow deployment to the Compaq IPAQ systems, because Virtual Machine on those devices only supported the 1.18 Java specification).

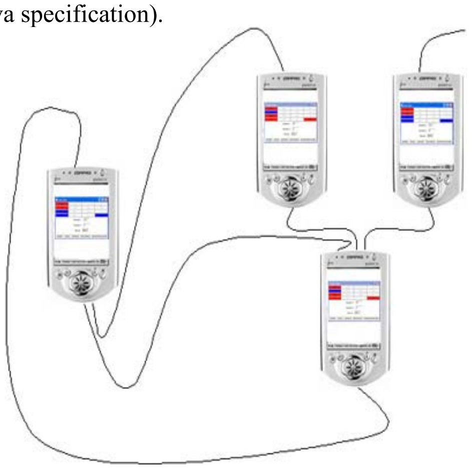

Figure 4. **The topology of our distributed calculator application. Each IPAQ can run the local calculator system and provide different math services to each remote calculator node, as well as local functionality to the user. The system was also deployed on some standalone Windows machines. The lines connecting the iPAQs are a logical representation of the actual wireless connectivity between them.**

The provided-services of each local device is populated with a text configuration file. The file is in the following format (‘#’ denotes a commented line):

```
# Services.conf Configuration File

# Component | Provided {1|0}

Gui 1

Add 1
```

```
#…

Sub 1
```

Each device ran the Java PRISM implementation we were given, and also included the extensions to PRISM that were described above.

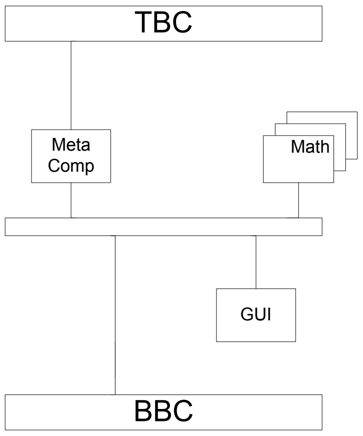

Figure 5. **The system architecture of each local distributed calculator. Note that this device receives incoming connections from other devices through the Bottom Border Connector, which we have labeled, "BBC". Also note that the Top Border Connector (TBC) is the main out bound form of communication for the device.**

We stored 4 main pieces of device meta-architecture information within each Meta Component's Device Table:

1) CPU Speed-a rating between 0 and 100 that rates the speed of a device's CPU
2) Network Throughput-a count of all the outbound messages delivered divided by the total number of messages sent
3) The IP Address of the device
4) The service port of the device

Table 2. Meta-architecture information stored in Device Table by each distributed calculator application

Device ID	IP Address	Service Port	CPU Speed	Network Throughput
D1	**127.99.92.225**	9009	85	35%
D2	**127.0.0.1**	9009	99	65%
D3	**23.23.33.222**	9009	91	75%
D4	**253.99.23.22**	9001	32	22%

Each distributed calculator also stored a list of provided math services in the Service Table. With this list of provided services, each device was able to map which services were available locally and which were available remotely. Then, if the device needed to consume math services that were not provided locally on its system because of a user request from the GUI component, and there were multiple remote devices that provided the same required service, then the requesting device could use its meta-architecture information about the other devices as a sort of heuristic to determine which devices to connect to, in order to consume the required math services. (For example, connect to the device with the maximum CPU Speed and/or network throughput).

4. CONCLUSION

We have presented an extension to the PRISM middleware that allows for management of device synchronization, management and consumption of local and remote device services, and the ability to reconnect lost peers. Our contribution allows PRISM to become fault tolerant for disconnected devices, which make re-entry into the network by "***remembering***" the services that each disconnected device provided, and a set of meta-level information regarding the disconnected device which included data for *Network Throughput*, a unique *Device ID,* a resource location *IP Address* and *CPU Speed*.

We have presented a detailed description of our sample application - a distributed calculator system that makes use of PRISM - and our extensions.

ACKNOWLEDGEMENT

Chris Mattmann is a Master of Science Candidate, with an emphasis in Multimedia and Creative Technologies in the Computer Science Department at the University of Southern California. His interestes are in multiagent systems, computer graphics and software architectures. He can be reached at mattmann@usc.edu.

Bilal Shaw is a Master of Science Candidate with an emphasis in Theory in the Computer Science Department at the University of Southern California. His interests are in experimental and theoretical algorithmic self-assembly and software architectures. He can be reached at bilalsha@usc.edu.

REFERENCES

[1] Marija Mikic-Rakic and Nenad Medvidovic. "Architecture-Level Support for Software Component Deployment in Resource Constrained Environments". *First International IFIP/ACM Working Conference on Component Deployment* (to appear). Berlin, Germany, June 2002.

[2] Marija Mikic-Rakic and Nenad Medvidovic. "Middleware for Software Architecture-Based Development in Distributed, Mobile, and Resource-Constrained Environments". Submitted.

[3] Nenad Medvidovic and Marija Mikic-Rakic. "Architectural Support for Programming-in-the-Many." Submitted. Available as *Technical Report, USC-CSE-2001-506*, University of Southern California, October 2001.

[4] Nenad Medvidovic, 'Formal Definition of the Chrion-2 Software Architectural Style', Technical Report UCI-ICS-95-24, University of California, Irvine, Irvine, CA 92717-3425, (1995).

[5] Nenad Medvidovic and Marija Mikic-Rakic. "Programming-in-the-Many: A Software Engineering Paradigm for the 21st Century." *Workshop on New Visions for Software Design and Productivity*: *Research and Applications*, Nashville, Tennessee, December 2001.